A Secure Software Engineering Perspective

Arun Mishra[1] and Arati M Dixit[2]
[1] Computer Engineering Department, Defence Institute of Advanced Technology, Pune, India
[2] Computer Engineering Department, Defence Institute of Advanced Technology, Pune, India
Email: arundoes@yahoo.co.in, adixit98@gmail.com

***Abstract* - Software vulnerabilities are the prime cause for the cyber attacks and potential misuse of software applications. The vulnerabilities are mostly due to unsecure system architecture, software development language and design issues. Generally software development practice does not address these issues due to time-budget constraints and conflicting needs. This ultimately results in software development, where security is a major concern, remains mainly unnoticed. Secure software engineering by and large refers to the process of software security. The software security essentially focuses on developing the secure software, which generally depends on system architecture and software security assurance against the possible vulnerabilities. To address these issues, in this paper, a survey is reported as a state of art work in the areas of secure system architecture, buffer overflow attacks and confinement.**

***Index Terms* - Secure Software Architecture, Verification, Buffer Overflow, Confinement.**

I. Introduction

Secure software engineering is the modus operandi of engineering software so that it persists to function appropriately under malicious attack. Today's most prevalent and widely discussed attacks exploit architectural, code-level, information leaks from side channel and software verification flaws. Malicious intruders can hack systems by exploiting these software defects. Thus it becomes essential for the software engineering community to think about these software vulnerabilities consecutively to identify and understand common threats.

The *Secure Software Architecture* can be achieved by imposing access control and security kernel mechanism [1]. The term access control applies only to subjects and objects within the system, and does not allow access to the system by outsiders. To implement access control, a system requires some built-in support to enforce the layering and proper use of the interfaces. Access control can be implemented at each layer with the help of access polices [2]. Policies consist of a precise set of rules for determining authorization as a basis for making access control decisions for individual layer. The security kernel approach [1] is a methodology based on the theory that in a large operating system (OS), a relatively small fraction of the software is responsible for security. By restructuring the OS, all of the security-relevant software is segregated into a trusted kernel of an OS, to avoid the security problems inherent in conventional designs. The section 2 discusses the various aspects of secure software architecture. The *Buffer Overflow* [3] vulnerability occurs when a program writes more data to a buffer against the actual allocated buffer. Buffer overflow vulnerabilities are very common and very easy to exploit. The injected attack code runs with the privileges of the vulnerable program and allows the attacker to self sustain and proceed without external help. The buffer overflow vulnerability is explored in the section 3.The *Confinement* [4] can be useful during software execution so that it cannot transmit information to any other program except its caller. This leakage can happen through a call on a program with memory, through misuse of storage facilities provided by the supervisor, or through channels intended for other uses onto which the information is encoded. The activities in the area of confinement are reported in section 4.

II. Secure Software Architecture

The secure software architecture theme is based on the concepts of increasing security through abstraction and decomposition of concerns i.e. separate security kernel [5]. For secure software architecture we focus on general-purpose OS. It is essential that the architecture of the system support its security-relevant aspects so that large system sections can be claimed as secure.

At the architecture level, the focus is to addresses internal security controls that are implemented within the hardware and software components of the system. The components within the system are of two types [5]: those accountable for preserving the security of the system, and all others. Separating the two types of components is an imaginary state line called the *security perimeter*. The typical components inside the security perimeter are the OS and computer hardware. The components outside the perimeter are user programs, printers, peripheral resources and the attributes that the system controls and protects. A well-defined interface across the safety perimeter is essential and compulsory for the security-pertinent components. The security architecture is discussed in details in the succeeding section which describes the aspects of the system development process through which adherence to the security requirements is assured.

A. Secure System Architectures

The conventional decomposition of a computer system shows hardware, an OS, and applications programs [5], as in Fig. 1. There may be numerous requests running concurrently and independently. The interface among a pair of layers depicts the functions in the lower layer that are accessible to the upper layer. There are two important interfaces: the system boundary and the security perimeter. The OS, mutually with the hardware, guarantees that the security perimeter interface is accessed only in agreement with the

DOI: 03.LSCS.2013.1.582

rules or policies of that interface. In order for the OS to force constraints on the applications successfully, the OS must have at least two states: privileged, and unprivileged. The privileged mode is also described as an executive, system, kernel, or supervisor mode; and the unprivileged mode is referred as user, application, or problem mode. When the machine is running in privileged mode, software can affect any machine instruction and can access any location in memory. In unprivileged mode, software is prevented from executing certain instructions or accessing memory in a way that could cause damage to the privileged software or other processes.

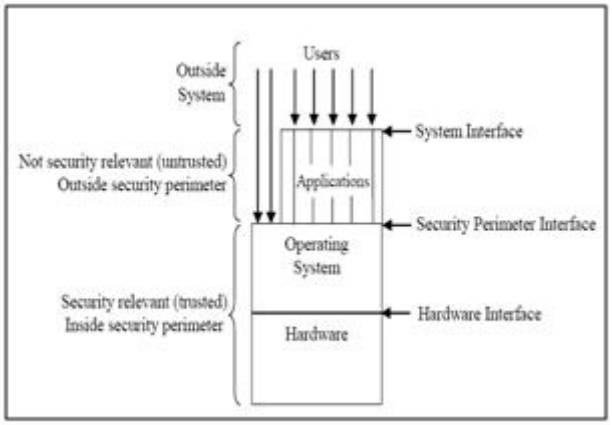


Fig. 1. Generic computer system architecture [5]

To implement the privileged and unprivileged states, there are two major concerns [6]:

1. Authentication
2. Security Policies.

1. Authentication

In order for a system to make significant decisions concerning whether a user should be permitted to access a system file, the system must have a means to recognize each user. A unique identifier must be linked with each user. The act of associating a user (or more accurately, a program running on behalf of a user) with a unique identifier is defined as an *authentication*. All actions within a system can be viewed as sequence of operations on objects. Generally an object is considered as a file, but otherwise anything that holds data may be an object, including memory, directories, queues, inter-process messages, network packets, input/ output (I/O) devices. The active entities that can access or influence objects are called subjects. At a high level of abstraction, users are subjects; but within the system, a subject is frequently considered to be a process, job, or task, operating on behalf of (and as a surrogate for) the user.

The primary purpose for security mechanisms in a system is access control, which consists of three tasks [5]:

1. Authorization to determine whether a subject is entitled to have access to particular objects.
2. Determination of the access rights which focuses on *read, write, execute, delete, and append* access modes.
3. Enforcement of the access rights

Subjects grant access rights to objects. Generally, a subject that possesses the ability to modify the access rights of an object is considered to be the object's owner, although there may be multiple owners. Associated with each object is a set of security attributes used to assist determine authorization and access rights. To determine access control implementation, it becomes necessary to distinguish between the granting of access rights (which happens in advance) and exercising of rights (which happens at the time of access), because security violations do not occur until an improper access takes place.

2. Security Policy

In the real world, a security policy governs the access to documents or other information [2]. The policy consists of a specific set of rules for determining authorization as a basis for making access control decisions. Lack of a clear policy and not programming errors is a major reason for the flaws in the security controls of most of systems. Generally, the system obeys security properties, while people obey a security policy. All systems have defined security properties and policies based on it.

The security kernel approach to building a system is responsible for enforcing the security policy of the system. In separate security kernel approach for a large OS, a fairly small fraction of the software is accountable for security [5]. The OS can be restructured so that all of the security-relevant software is segregated into a trusted kernel. In most respects, the security kernel is a primitive OS which performs services on behalf of the OS. The security kernel must be suitably protected, and it must not be possible to bypass the kernel's access control checks. The kernel must be as small as possible so that its correctness is easy to verify. The security kernel in the Fig. 1, consists of a new layer of software inserted between the hardware and OS.

Access decisions specified by the policy are based on information in an abstract access control database. The access control database embodies the security state of the system and contains information such as security attributes and access rights. The database is dynamic, changing as subjects and objects are created or deleted, and as their rights are modified. A key requirement of the kernel is the control of each and every access from subject to object. Based on a set of strict principles that guide the design and development process, the security kernel approach can significantly increase the user's level of confidence in the correctness of the system's security controls in secure system architecture.

The two major principles [6] for design of the secure system architecture are:

1. Minimize and Isolate Security Controls

To achieve a high degree of confidence in the security of a system, the designer should minimize the size and complexity of the security-relevant parts of the system design. The key to minimizing the security-relevant parts of OS is to design the system to use only a small number of different types of security enforcement mechanisms, thereby forcing security-relevant actions to be taken in a few isolated sections.

2. Enforce Least Privilege

Closely related to the concept of isolating the security mechanisms is the principle of least privilege: subjects should

DOI: 03.LSCS.2013.1.582

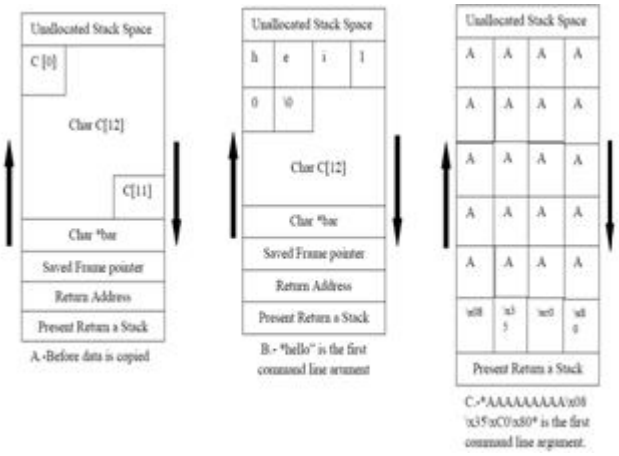
be given no more privilege than is necessary to enable them to do their jobs. In that way, the damage caused by erroneous or malicious software is limited.

III. Buffer Overflow Vulnerability

Buffer overflow vulnerabilities are very common and very easy to exploit. The injected attack code runs with the privileges of the vulnerable program and allows the attacker to self sustain and proceeds without external help. A survey on the Bugtraq security vulnerability mailing list [7] showed that approximately 2/3 of respondents felt that buffer overflows are the leading cause of security vulnerability and the remaining 1/3 of respondents identified *misconfiguration* as the leading cause of security vulnerability. Ken Thompson [8] the principal inventor of UNIX OS also acknowledged the buffer overflow vulnerabilities in his paper "Reflections on Trusting Trust". He has shown buffer overflows implication in the C compiler which is written in its native language.

Buffer Overflow occurs while writing data to a buffer which overruns the buffers boundary and overwrites adjacent memory. This is a special case of memory violation or corruption. Programs written in C language [9] are prone to buffer overflow attack. The buffer overflow is also known as stack smashing. In some cases buffer overflow may cause the program to crash or operate incorrectly.

A *Stack smashing* example is illustrated in Fig 2. The method for exploiting a stack based buffer overflow is to overwrite the function return address with a pointer to attacker controlled data [14] (usually on the stack itself). The Fig. 2(a) displays the stack before data is copied. The first command line argument 'hello' is represented on the stack space in Fig. 2(b). The hex representation of the first command line argument is represented on the stack space in Fig. 2(c).



(a) (b) (c)

Fig. 2. Memory allocation in Stack [14]

Consider the String Copy Function in program 1. When user supplies an argument larger than 11 bytes from the command line, foo() overwrites local stack data, the saved frame pointer, and most importantly, the return address as seen from Fig. 2(C).

Program 1. String Copy Function

```
#include <string.h>
void foo (char *bar)
{
        char C[12];
    // no bounds checking...
        strcpy(c, bar);
}
int main (int argc, char **argv)
{
foo(argv[1]);
}
```

Then the control will go to the attacker's code. If the above program had special privileges (e.g. the SUID bit set to run as the super user), then the attacker could use this vulnerability to gain super user privileges on the affected machine especially when the affected program is running with special privileges, or accepts data from un-trusted network hosts then the bug is potential security vulnerability. This happens in the above code as there is no array bound checking done explicitly. All the string manipulation functions such as *strcpy*, *strcat*, *sprint* etc are vulnerable to buffer overflow attack.

The major applications of UNIX OS vulnerable to buffer overflow attacks [11] are as follows:

- *Traceroute*
- *libc* libraries handle environment variables
- The most popular strategy often called '*return-to-libc*' is to modify the function's return address to point to a well known function such as a runtime library function or a system API.
- *csh* – C shell
- *ps* – used to display information about running processes.
- *Sendmail* - used to send and display mail messages
- *strcpy()* - string copy
- *strcat()* – string concatenation
- *gets()* – does not know the number of characters safely stored in the string passed to it.
- *sprintf()* – used to write formatted data to string

The Worms are created by using the Buffer overflow vulnerability. The buffer overflow vulnerability exploited by some of the famous worms can be enlisted as follows:

- The Morris worm [15] took control of the UNIX finger daemon fingerd by exploiting a stack buffer overflow.
- The Internet Security Systems (ISS) desktop agent BlackICE was exploited by the Witty worm [16] using stack based buffer overflow
- Microsoft's SQL server's stack buffer overflow exploited by the Slammer worm [17].
- Microsoft DCOM service's stack buffer overflow exploited by the Blaster worm [18].

A. Protection Scheme for Buffer Overflow

Buffer overflows are troublesome as they can go undetected during the development and testing of software applications. Common C and C++ compilers neither identify possible buffer overflow conditions at compilation time nor report buffer overflow exceptions at runtime. The possible protection mechanisms [12] are:

DOI: 03.LSCS.2013.1.582

a) Stack Buffer Overflow Detection to prevent redirection of the instruction pointer to malicious code.The Stack canaries or cookies can be utilized for the stack buffer overflow detection. Place a small integer value which is randomly chosen at program start, in memory just before the stack return pointer. Most buffer overflows overwrite memory from lower to higher memory addresses, so in order to overwrite the return pointer the canary value must also be overwritten. This value is checked to make sure it has not changed before a routine uses the return pointer on the stack. Push canary to the stack between the last local variable and the function's return address. If the cookie has been modified, program execution immediately stops. It prevents the execution of any kind of malicious code.

b) Malicious Code Execution Prevention from the stack without directly detecting the stack buffer overflow. Preventing stack buffer overflow exploitation is to enforce memory policy on stack memory region to disallow execution from the stack. Processors provide support for defining memory pages as non executable. Those pages can only be used for storing data and the processor will not run code stored in them. This is also called "Data Execution Prevention" mechanism. The OS marks the stack and pages as non executable. It prevents an attacker from running code on them using a buffer overflow. The newer processors and OS support this feature. The Intel, AMD, IA-64 processors support this feature. The OS like Windows XP service pack 2 and above as well as the Solaris 2.6 Linux kernel with added patches supports the feature of non-executable stacks.

B. Coding Practices

The progression of writing correct code is a costly proposition particularly in case of a language like C. The C language has error prone expressions [7] such as null terminated strings and a way of coding that favors performance over correctness. Some of the coding practices [10] can help in reduction of the buffer overflow problem with no guarantee of complete elimination of the problem. Some of the coding practices recommended [10] are:

1. A simple solution is to test the length of the input using *strlen()* and then allocate the memory. This is applicable for all string manipulation functions.
2. Use *fgets()* instead of *gets()* function
3. *Scanf()* can be used safely by specifying a width. (e.g. Format %10s will read less than or equal to 10 characters)
4. Use *snprintf()* instead of *sprintf()* which specifies the number of characters to be read.
5. Use *strncpy()* instead of *strcpy()*.
6. Compile the code using "-check_bounds" option which will perform array bounds checking.
7. Check each warning statement produced by the compiler and perform necessary corrections.

IV. The Confinement Predicament

The primary goal to protect the systems includes safeguarding:

- Data from unauthorized access or modification, and
- Programs from unauthorized execution.

Further, in a client-server application, there is a possibility of information leakage. The client depends on the server to provide services. The client may not trust a server and it is possible that the server may record the information with some malicious intentions. There is a possibility that the server may sell this information to the interested parties, causing damage to the client. This issue can be avoided by disallowing the server to record any information. Server can record information by writing it into an external file or communicate with other process, called collaborator. The *Confinement Problem* essentially can be defined as the main objective of the system to make sure that there is no way for the server to leak information to the collaborator.

Confining an arbitrary program does not mean that any program which works when free (i.e. not confined) will still work under confinement, but that any program, if confined, will be unable to leak data. A misbehaving program may well be trapped as a result of an attempt to escape [13]. Compilation of list of possible leaks as discussed in [4] facilitates the process of creation of some intuition in preparation for a more abstract description of confinement. A confined program must not support memory, thus avoiding conservation of information within itself. The rules that enable the confinement [4] are:

- A confined program remains focused on the intra- process calls, with a complete restriction to making calls to other programs.
- In a situation in which total isolation is not practical and a call have to be invoked for another program then, the called program must also be confined.
- In a scenario with a need of storage, preserved by the supervisor, a due care has to be taken to avoid any information leakage.
- Lawful channels used by the confined service,such as the invoice statements, may be avoided at the client side to prevent any information leakage.
- Circumvent covert channels, i.e. those not intended for information transfer should be handled carefully.

Thus the straightforward and undemanding principles of masking and enforcement are adequate to obstruct all legitimate and covert channels.

Conclusion

An overview of the existing techniques in secure software engineering environment is presented with a focus on four principle security components. The principle security components like Secure Software Architecture, buffer overflows and confinement are discussed. The main aim of this paper is to generate awareness about the vulnerabilities and system exploitation in software practitioner community. The implementation of techniques to block the ambiguity due to the vulnerabilities is still an exclusive affair.

References

[1] Ames, S. R., Jr.; Gasser, M.; and Schell, R. R. 1983. "Security

DOI: 03.LSCS.2013.1.582

Kernel Design and Implementation: An Introduction." Computer 16(7):14-22. Reprinted in Advances in Computer System Security, vol. 2, ed. R. Turn, pp. 170-77, Artech House.

[2] Clark, D. D., and Wilson, D. R. 1987. "A Comparison of Commercial and Military Computer Security Policies." In Proceedings of the 1987 Symposium on Security and Privacy, pp. 184-95. Washington, D.C.: IEEE Computer Society.

[3] Steve Bellovin. "Buffer Overflows and Remote Root Exploits". Personal Communications,October 1999.

[4] Butler W. Lampson. 1973. "A note on the confinement problem". Communication , ACM 16, 10 (October 1973), 613-615.

[5] Morrie Gasser. 1988. "Building a Secure Computer System". Van Nostrand Reinhold Co., New York, NY, USA.

[6] Jim Alves-Foss, W. Scott Harrison, Paul Oman and Carol Taylor. "The MILS Architecture for High-Assurance Embedded Systems". International Journal of Embedded Systems at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.6810&rep=rep1&type=pdf.

[7] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie and Jonathan Walpole, "Buffer overflows: Attacks and Defences for the Vulnerability of the Decade", DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings Date of Conference: 2000, vol.2, pp. 119 - 129.

[8] Ken Thompson, "Reflections on Trusting Trust", Communications of the ACM, August 1984, Volume 27, Number 8, pp. 761-763.

[9] Kernighan, B.W., and Ritchie, D.M. "The C Programming Language". Prentice-Hall, Englewood Cliffs, N.J., 1978.

[10] EUROSEC GmbH Chiffriertechnik & Sicherheit, "Secure Programming in C/C++", ver 1.0, July 2005, http://www.secologic.org/downloads/c/051207_EUROSEC_Draft_Whitepaper_Secure_C_Programming.pdf

[11] Rajib K. Mitra, (1998), "UNIX Security", [online], http://www.spy.net/~jeeb/unix_security.html

[12] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In 7th USENIX Security Conference, pages 63-77, San Antonio, TX, January 1998.

[13] Lampson, B.W., "Dynamic protection structures", Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Niontvale, N.J., pp. 27-38.

[14] "Stack buffer overflow " http://wikipedia.org/wiki/Stack_buffer_overflow.htm (Oct. 08, 2012)

[15] Bob Page,(1988, November 7), "A Report on the Internet Worm", [online], http://www.ee.ryerson.ca/~elf/hack/iworm.html

[16] "Witty Worm targets BlackICE PC Protection systems (ICQ_Witty_Worm)", [online], http://www.iss.net/security_center/reference/vuln/ICQ_Witty_Worm.htm

[17] Paul Boutin, (2003 July), "Slammed! An inside view of the worm that crashed the Internet in 15 minutes", Issue 11.07, [online], http://www.wired.com/wired/archive/11.07/slammer.html (Oct. 08, 2012)

[18] "Blaster (computer worm)", (2012 December 26), [online], http://en.wikipedia.org/wiki/Blaster_worm

DOI: 03.LSCS.2013.1.582